MinMax-Profiles: A Unifying View of Common Intervals, Nested Common Intervals and Conserved Intervals of K Permutations

Irena Rusu[1]
L.I.N.A., UMR 6241, Université de Nantes, 2 rue de la Houssinière,
BP 92208, 44322 Nantes, France

Abstract

Common intervals of K permutations over the same set of n elements were firstly investigated by T. Uno and M.Yagiura (Algorithmica, 26:290:309, 2000), who proposed an efficient algorithm to find common intervals when $K = 2$. Several particular classes of intervals have been defined since then, *e.g.* conserved intervals and nested common intervals, with applications mainly in genome comparison. Each such class, including common intervals, led to the development of a specific algorithmic approach for $K = 2$, and - except for nested common intervals - for its extension to an arbitrary K.

In this paper, we propose a common and efficient algorithmic framework for finding different types of common intervals in a set $\mathcal{P}$ of K permutations, with arbitrary K. Our generic algorithm is based on a global representation of the information stored in $\mathcal{P}$, called the *MinMax*-profile of $\mathcal{P}$, and an efficient data structure, called an *LR*-stack, that we introduce here. We show that common intervals (and their subclass of irreducible common intervals), nested common intervals (and their subclass of maximal nested common intervals) as well as conserved intervals (and their subclass of irreducible conserved intervals) may be obtained by appropriately setting the parameters of our algorithm in each case. All the resulting algorithms run in $O(Kn + N)$-time and need $O(n)$ additional space, where N is the number of solutions. The algorithms for nested common intervals and maximal nested common intervals are new for $K > 2$, in the sense that no other algorithm has been given so far to solve the problem with the same complexity, or better. The other algorithms are as efficient as the best known algorithms.

Keywords: permutation, algorithm, common intervals, conserved intervals, nested common intervals

1 Introduction

Common, conserved and nested common intervals of two or more permutations have been defined and studied in the context of genome comparison. Under the assumption that a set of genes occurring in neighboring locations within several genomes represent functionally related genes [11, 17, 21], common intervals and their subclasses are used to represent such conserved regions, thus helping for instance to detect clusters of functionally related genes [18, 22], to compute similarity measures between genomes [6, 3] or to predict protein functions [15, 24]. Further motivations and details may be found in the papers introducing these intervals, that we cite below.

In these applications, genomes may be represented either as permutations, when they do not contain duplicated genes, or as sequences. In sequences, duplicated genes usually play similar roles and yield the interval search more complex [10, 20], but sometimes they are appropriately matched and renumbered so as to obtain permutations [9, 2].

We focus here on the case of permutations. Efficient algorithms exist for finding common and conserved intervals in K permutations ($K \geq 2$), as well as for finding irreducible common and irreducible conserved intervals [23, 12, 5, 6]. Nested common intervals and maximal nested common intervals have been studied more recently [14], and efficient algorithms exist only for the case of two permutations [8].

[1]Irena.Rusu@univ-nantes.fr

Surprisingly enough, whereas all these classes are subclasses of common intervals, each of them has generated a different analysis, and a different approach to obtain search algorithms. Among these approaches, interval generators [5] have been shown to extend from common intervals to conserved intervals [19], but this extension is not easily generalizable to other subclasses of common intervals.

The approach we present in this paper exploits the natural idea that an efficient algorithm for common intervals should possibly be turned into an efficient algorithm for a subclass of common intervals by conveniently setting some parameters so as to filter the members of the subclass among all common intervals. It also chooses a different viewpoint with respect to the information to be considered. Instead of searching intervals directly in the permutations, it first extracts the helpful information from the permutations, focusing on each pair $(t, t+1)$ of successive values in $\{1, 2, \ldots, n\}$ and defining the so-called *MinMax*-profile of the permutations. Then, it progressively computes the set of interval candidates, but outputs them only after a filtering procedure selects the suitable ones.

The organization of the paper is as follows. In Section 2, we present the main definitions and the problem statement. Then, in Section 3, we introduce the abstract data structure on which strongly relies our main algorithm called *LR*-Search, also described in this section. The complexity issues are discussed in Section 4. In Section 5 we give the specific settings of our algorithm for common, nested common and conserved intervals, and prove the correction in each case. In Section 6 we show how to further modify the algorithm so as to deal with even smaller subclasses. Section 7 is the conclusion.

2 Generalities

For each positive integer u, let $[u] := \{1, 2, \ldots, u\}$. Let $\mathcal{P} := \{P_1, P_2, \ldots, P_K\}$ be a set of permutations over $[n]$, with $n > 0$ and integer. The i-th element of P_k is denoted p_i^k, for all $i \in [n]$. The *interval* $[i, j]$ of P_k, defined only for $1 \leq i < j \leq n$, is the set $\{p_i^k, p_{i+1}^k, \ldots, p_j^k\}$, and it is denoted $(i..j)$ when P_k is the identity permutation $Id_n := (1\,2\,\ldots\,n)$. Then $(i..j) = \{i, i+1, \ldots, j\}$. For an interval $[i, j]$ of P_k, we also say that P_k is *delimited* by p_i^k and p_j^k, or equivalently that p_i^k and p_j^k are the *delimiters* of $[i, j]$ on P_k. Furthermore, let $P_k^{-1} : [n] \to [n]$ be the function, easily computable in linear time, that associates with every element of P_k its position in P_k. We now define common intervals:

Definition 1. [23] A *common interval* of $\mathcal{P}$ is a set of integers that is an interval of each P_k, $k \in [K]$.

Nested common intervals are then defined as follows:

Definition 2. [14] A *nested common interval* (or *nested interval* for short) of $\mathcal{P}$ is a common interval I of $\mathcal{P}$ that either satisfies $|I| = 2$, or contains a nested interval of cardinality $|I| - 1$.

Example 1. Let $P_1 = Id_7$ and $P_2 = (7\,2\,1\,3\,6\,4\,5)$. Then the common intervals of $\mathcal{P} = \{P_1, P_2\}$ are $(1..2), (1..3), (1..6), (1..7), (3..6), (4..5)$ and $(4..6)$, whereas its nested intervals are $(1..2), (1..3), (4..5)$, $(4..6)$ and $(3..6)$.

With the aim of introducing conserved intervals, define now a *signed* permutation as a permutation P associated with a boolean vector $sign_P$ that provides a $+$ or $-$ sign for every element of P. Then $sign_P[i]$ is the sign of the integer i in P. Note that, even in a signed permutation, the elements are positive integers. An element of P is called *positive* or *negative* if its associated sign is respectively $+$ or $-$. A permutation is then a signed permutation containing only positive elements. Conserved intervals need to assume that $p_1^k = 1$ (positive) and $p_n^k = n$ (positive), for each $k \in [K]$.

Definition 3. [6] Let $\mathcal{P} := \{P_1, P_2, \ldots, P_K\}$ be a set of signed permutations over $[n]$, with $p_1^k = 1$ (positive) and $p_n^k = n$ (positive), for each $k \in [K]$. A *conserved interval* of $\mathcal{P}$ is a common interval of $\mathcal{P}$ (ignoring the signs) delimited on P_k by the values a_k (left) and b_k (right), for each $k \in [K]$, such that exactly one of the following conditions holds for each k:

(1) $a_k = a_1$ with the same sign and $b_k = b_1$ with the same sign.

(2) $a_k = b_1$ with different signs and $b_k = a_1$ with different signs.

Example 2. Let $P_1 = Id_7$ and $P_2 = (1\,3\,2\,6\,4\,5\,7)$ with $sign_{P_2} = [+,-,-,-,-,+,+]$, meaning that $1, 6$ and 7 are positive, whereas $2, 3, 4$ and 5 are negative in P_2. Then the conserved intervals of $\mathcal{P} = \{P_1, P_2\}$ are $(1..7)$ and $(2..3)$, whereas its common intervals (ignoring signs) are $(1..3), (1..6), (1..7), (2..3), (2..6), (2..7), (4..5), (4..6), (4..7)$.

In the following problem, $\mathcal{C}$ refers to a class of common intervals, *e.g.* common, nested or conserved.

$\mathcal{C}$-INTERVAL SEARCHING PROBLEM (abbreviated $\mathcal{C}$-ISP)
Input: A set $\mathcal{P} = \{P_1, P_2, \ldots, P_K\}$ of signed permutations over $[n]$, satisfying the conditions required by the definition of $\mathcal{C}$.
Requires: Give an efficient algorithm to output without redundancy all $\mathcal{C}$-intervals of $\mathcal{P}$.

In all cases, we may assume without loss of generality that $P_1 = Id_n$, by appropriately renumbering P_1. The other permutations must be renumbered accordingly.

In this paper, we propose a common efficient algorithm to solve $\mathcal{C}$-ISP when $\mathcal{C}$ stands for common, nested, conserved intervals as well as for their respective subclasses of irreducible common, maximal nested and irreducible conserved intervals (defined in Section 6). For common and conserved intervals (irreducible or not), efficient algorithms have been proposed so far, developping different and sometimes very complex approaches. For nested and maximal nested intervals, efficient algorithms exist for the case $K = 2$. We solve here the case of an arbitrary K. Our approach is common for all classes, up to the filtering of the intervals in $\mathcal{C}$ among all common intervals.

3 Main Algorithm

Our *LR*-Search algorithm is based on two main ideas. First, it gathers information from $\mathcal{P}$ that it stores as *anonymous* constraints on each pair $(t, t+1)$ of successive values, since during the algorithm it is useless to know the source of each constraint. Second, it fills in a data structure that allows us to find *all* common intervals with provided constraints if we need, but an additional Filter procedure is called to choose and to output only the intervals in the precise class $\mathcal{C}$ for which the algorithm is designed.

The algorithm uses a specific data structure that we call an *LR*-stack. The candidates for the left (respectively right) endpoint of a common interval are stored in the L (respectively R) part of the *LR*-stack. At each step of the algorithm, the *LR*-stack is updated, the solutions just found are output, and the *LR*-stack is passed down to the next step.

3.1 The *LR*-stack

The *LR*-stack is an abstract data structure, whose efficient implementation is discussed in Section 4.

Definition 4. An *LR-stack* for an ordered set Σ is a 5-tuple $(L, R, SL, SR, R^{\top})$ such that:

- L, R are stacks, each of them containing distinct elements from Σ in either increasing or decreasing order (from top to bottom). The *first* element of a stack is its top, the *last* one is its bottom.
- $SL, SR \subset \Sigma$ respectively represent the set of elements on L and R.
- $R^{\top} : SL \to SR$ is an injection that associates with each a from SL a pointer to an element on R such that $R^{\top}(a)$ is before $R^{\top}(a')$ on R iff a is before a' on L.

According to the increasing (notation +) or decreasing (notation -) order of the elements on L and R from top to bottom, an *LR*-stack may be of one of the four types $L^+R^+, L^-R^-, L^+R^-, L^-R^+$.

Remark 1. We assume that each of the stacks L, R admits the classical operations $pop, push$, and that their elements may be read without removing them. In particular, the function *top*() returns the first element of the stack, without removing it, and the function *next*(u) returns the element immediately following u on the stack containing u, if such an element exists.

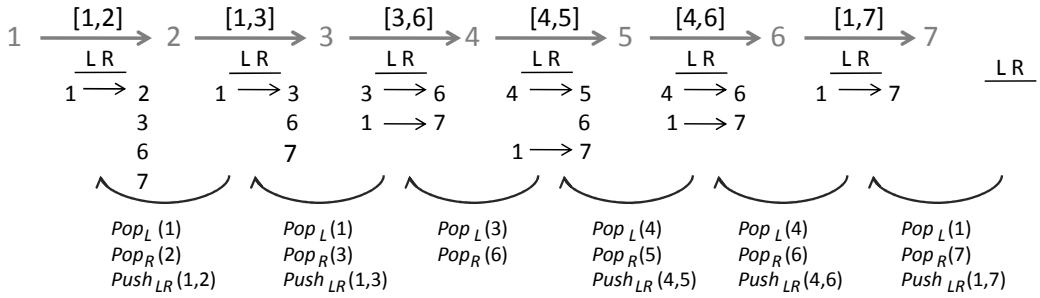

Figure 1: *MinMax*-profile of $\mathcal{P} = \{Id_7, (7\,2\,1\,3\,6\,4\,5)\}$ with basic bounding functions $b(t) = m_t$ and $B(t) = M_t$, and execution of the *LR*-Search algorithm. The stack is initially empty. For each pair $(t, t+1)$ with $t = n-1, n-2, \ldots, 1$, the corresponding *LR*-stack (below the pair) is obtained from the preceding *LR*-stack (on its right) by performing the sequence of operations written below the arrow linking the two *LR*-stacks.

We further denote, for each $a \in SL$ and with $a' = next(a)$, assuming that $next(a)$ exists:

$$Set_R(a) = \{b \in SR \mid b \text{ is located on } R \text{ between } R^\top(a) \text{ included and } R^\top(a') \text{ excluded}\}$$

When $next(a)$ does not exist, $Set_R(a)$ contains all elements between $R^\top(a)$ included and the bottom of R included. Then $R^\top(a)$ is the first (i.e. closest to the top) element of $Set_R(a)$ on R.

The following operations on the LR-stack are particularly useful. Note that they do not affect the properties of an *LR*-stack. Sets $Set_R()$ are assumed to be updated without further specification whenever the pointers $R^\top()$ change. Say that a' is *L-blocking* for a, with $a' \neq a$, if a cannot be pushed on L when a' is already on L (because of the increasing/decreasing order of elements on L), and similarly for R.

- $Pop_L(a)$, for some $a \in \Sigma$: pop successively from L all elements that are L-blocking for a, push a on L iff at least one L-blocking element has been found and a is not already on L, and define $R^\top(top(L))$ as $top(R)$. At the end, either a is not on L and no L-blocking element exists for a, or a is on the top of L and $R^\top(a)$ is a pointer to the top of R.
- $Pop_R(b)$, for some $b \in \Sigma$: pop successively from R all elements that are R-blocking for b, update all pointers $R^\top()$ and successively pop from L all the elements a with $R^\top(a) = nil$. At the end, either b is not on R and no R-blocking element exists for b, or b is on the top of R.
- $Push_{LR}(a, b)$, for some $a, b \in \Sigma$ (performed when no L-blocking element exists for a and no R-blocking element exists for b): push a on L iff a is not already on the top of L, push b on R iff b is not already on the top of R, and let $R^\top(top(L))$ be defined as $top(R)$.
- $Find_L(b)$, for some $b \in \Sigma$: return the element a of *SL* such that $b \in Set_R(a)$.

Example 3. In Figure 1, consider the L^-R^+-stack below the pair $(4, 5)$. Here, $Set_R(4) = \{5, 6\}$ and $Set_R(1) = \{7\}$. The instruction $Pop_L(3)$ discards 4 from L, and pushes 3 instead, also defining $R^\top(3) = 5$. Next, the instruction $Pop_R(6)$ discards 5 from R. The resulting *LR*-stack is represented below the pair $(3, 4)$.

3.2 The *LR*-Search algorithm

Let $\mathcal{P}$ be a set of K signed permutations over $[n]$. Recall that $P_1 = Id_n$. In the following definitions, $2 \leq k \leq K$ and $1 \leq t \leq n-1$:

Algorithm 1 The *LR*-Search algorithm

Input: Set $\mathcal{P}$ of signed permutations over $[n]$, bounding functions b and B, Filter procedure
Output: All common intervals $(t..x)$ of $\mathcal{P}$ with $x \in Set_R(t)$, filtered by Filter

1: Compute m_t^k, m_t and b_t with $2 \leq k \leq K$ and $t \in [n-1]$
2: Compute M_t^k, M_t and B_t with $2 \leq k \leq K$ and $t \in [n-1]$
3: Initialize an L^-R^+-stack with empty stacks L, R
4: **for** t from $n-1$ to 1 **do**
5: $Pop_L(b_t)$ // *discard from L all candidates larger than b_t and push b_t instead*
6: $Pop_R(B_t)$ // *discard from R all candidates smaller than B_t*
7: **if** $B_t = t+1$ **then**
8: $Push_{LR}(b_t, t+1)$ // *$t+1$ is a new right candidate, suitable for each a on L*
9: **end if**
10: Call Filter to choose a subset of intervals $(t..x)$ with $x \in Set_R(t)$
11: **end for**

$$m_t^k := \min\{p_i^k \mid p_i^k \text{ is located between } t \text{ (included) and } t+1 \text{ (included) on } P_k\}$$
$$m_t := \min\{m_t^k \mid 2 \leq k \leq K\}$$
$$M_t^k := \max\{p_i^k \mid p_i^k \text{ is located between } t \text{ (included) and } t+1 \text{ (included) on } P_k\}$$
$$M_t := \max\{M_t^k \mid 2 \leq k \leq K\}$$

Notice that, for each $k \in [K]$,

$$m_t \leq m_t^k \leq t < t+1 \leq M_t^k \leq M_t.$$

Now, let the *bounding* functions $b, B : [n] \to [n]$ be two functions such that $b(t) \leq m_t$ and $B(t) \geq M_t$. Denote $b_t := b(t)$ and $B_t := B(t)$ for all $t \in [n-1]$.

Definition 5. Let $\mathcal{P}$ be a set of permutations on $[n]$. Then the *MinMax-profile* of $\mathcal{P}$ with respect to b and B is the set of pairs $[b_t, B_t]$, $t \in [n-1]$.

The *LR*-Search algorithm (See Algorithm 1) works as follows. Each pair $(t, t+1)$, $1 \leq t \leq n-1$, is associated with the couple $[b_t, B_t]$, which means intuitively that every common interval $(a..c)$ of $\mathcal{P}$ that contains t and $t+1$ should satisfy $a \leq b_t < B_t \leq c$. The L^-R^+-stack, initially empty, stores on L (respectively on R) the candidates for the left endpoint a (respectively right endpoint c) of a common interval $(a..c)$, and links together the left and right candidates to form suitable pairs. Intuitively, a suitable pair is a pair of endpoints for which no disagreement has been found yet. In the *LR*-stack, a suitable pair is a pair $(a, c) \in SL \times SR$ such that $a \leq Find_L(c)$. Equivalently, right candidate c could form common intervals with $Find_L(c)$ as well as with all left candidates following $Find_L(c)$ on L.

Remark 2. It is important to notice that, by definition, in an L^-R^+-stack, a' is L-blocking for a iff $a' > a$, and b' is R-blocking for b iff $b' < b$.

In its most general form, the algorithm outputs all common intervals without any redundancy. However, in order to insure its efficiency even for strict subclasses $\mathcal{C}$ of common intervals, it has two types of parameters allowing us to filter intervals during the search (bounding functions achieve that) and during the output step (Filter procedure achieves that). Then, *LR*-Search considers (step 4) every pair $(t, t+1)$ using decreasing values of t, so as to output the intervals with same left endpoint altogether in the same step (increasing values of t would produce the intervals with same right endpoint in the same step). Operations $Pop_L(b_t), Pop_R(B_t)$ (steps 5, 6) successively allow us to discard left candidates a that do not satisfy $a \leq b_t$ (respectively right candidates c that do not satisfy $B_t \leq c$), and to update the suitable pairs accordingly. Steps 7-9 identify the cases where $t+1$ may be added to R, namely when

$B_t = t+1$ (*Push*$_{LR}$ tests whether $t+1$ is already on R). Finally, in step 10 suitable intervals $(t..x)$ are necessarily common intervals since further constraints on pairs $(t', t'+1)$, $t' < t$, will not affect $(t..x)$. The Filter procedure is then called to filter among all common intervals found here those that belong to some specific subclass.

Example 4. Figure 1 shows the application of the *LR*-Search algorithm on the set $\mathcal{P} = \{P_1, P_2\}$ of permutations, where $P_1 = Id_7$ and $P_2 = (7\,2\,1\,3\,6\,4\,5)$. In this case, we defined $b_t = m_t = m_t^2$ and $B_t = M_t = M_t^2$. The constraints $[b_t, B_t]$ are written above the arrow representing the pair $(t, t+1)$, for all $t \leq n-1$, so that the upper part of the figure represents the *MinMax*-profile of $\mathcal{P}$. The first step corresponds to $t = 6$, and consists in performing operations *Pop*$_L(1)$, *Pop*$_R(7)$ and *Push*$_{LR}(1, 7)$ on the initially empty stack. Thus the first two operations have no effect, whereas the third one pushes 1 on L, 7 on R and defines $R^\top(1)$ as a pointer to 7. The next step takes the current state of the *LR*-stack and performs *Pop*$_L(4)$, *Pop*$_R(6)$ and *Push*$_{LR}(4, 6)$ to obtain the *LR*-stack below the pair $(5, 6)$. The first common intervals are output when $t = 4$, namely $(4..5), (4..6)$.

Remark 3. In the rest of the paper, the notation **for**$_t$ concerns the execution of the **for** loop in *LR*-Search for some fixed t. Similar notations will be used for the loops in the Filter procedures given subsequently. When these notations are not confusing, we use them without any further specification.

Let *Set*$_R^t(a)$ be the value of *Set*$_R(a)$ at the end of step 9 in **for**$_t$, for each a on L. Let *Pairs*$_t$ be the set of pairs $(t..x)$ with $x \in$ *Set*$_R^t(t)$.

Theorem 1. *Assuming the Filter procedure does not change the state of the LR-stack, the set A defined as $A := \cup_{1 \leq t < n}$ Pairs$_t$ computed by LR-Search is the set of all common intervals $(t..x)$ of $\mathcal{P}$ (ignoring the signs) satisfying $t = b_t$ and $x = B_{x-1}$. Moreover, for all these intervals we have*

$$b_t = \min\{b_w \,|\, t \leq w \leq x-1\}$$

$$B_{x-1} = \max\{B_w \,|\, t \leq w \leq x-1\}.$$

We first prove the following result. Notations *min*(I) and *max*(I) for an interval I of P respectively denote the minimum and maximum value in I.

Claim 1. *Let t, $1 \leq t < n$. After the execution of step 9 in* **for**$_t$*, we have $x \in$ Set$_R^t(a)$ iff t, x and a satisfy the three conditions below:*

(a) $a = \min\{b_w \,|\, t \leq w \leq x-1\}$

(b) $x = B_{x-1} = \max\{B_w \,|\, t \leq w \leq x-1\}$

(c) *for each k, $2 \leq k \leq K$, the interval I_k of P_k delimited by the leftmost and rightmost values among $t, t+1, \ldots, x$ satisfies min$(I_k) \geq a$ and max$(I_k) = x$.*

Proof of Claim 1. We use induction on t, with decreasing values. Let $t = n-1$ and notice that the *LR*-stack is empty up to step 7 in **for**$_{n-1}$.

Proof of "⇒:" The only operation that can affect the empty stack is *Push*$_{LR}(b_{n-1}, n)$, performed only when $B_{n-1} = n$. Then *Set*$_R^{n-1}(b_{n-1}) = \{n\}$. Thus $a = b_{n-1}, x = n$ and affirmations $(a), (b)$ are verified. Now, the interval I_k delimited by $n-1$ and n satisfies the required conditions since *min*$(I_k) = m_{n-1} \geq b_{n-1} = a$ and *max*$(I_k) = n = x$.
Proof of "⇐:" Let a, x and I_k be defined as in (a)-(c). Because of $t = n-1$, together with $x \leq n$ (since x is an element of $[n]$) and with $t \leq w \leq x-1$, we deduce that $x = n$ and $w = n-1$. Thus $a = b_{n-1}$. By affirmation (b), $B_{n-1} = n$. In conclusion, we have $b_{n-1} = a$ and $B_{n-1} = n$ thus the conditions in step 7 are fulfilled. Then a is pushed on L, n is pushed on R and we are done.

Assume now the claim is true for $t+1$, where $t+1 \leq n-1$, and let us prove it for t. For that, we denote $a' := \min\{b_w \,|\, t+1 \leq w \leq x-1\}$.

Proof of "⇒:" With $x \in$ *Set*$_R^t(a)$, three cases are possible:

(i) $x \in \mathit{Set}_R^{t+1}(a)$. In this case, during **for**$_t$ a is not discarded by $\mathit{Pop}_L(b_t)$, thus (1) $a \leq b_t$. Moreover, by the inductive hypothesis (affirmation (a) for $t+1$), $x \in \mathit{Set}_R^{t+1}(a)$ implies that (2) $a = \min\{b_w \,|\, t+1 \leq w \leq x-1\} = a'$. Then by (1) and (2):

$$\min\{b_w \,|\, t \leq w \leq x-1\} = \min\{b_t, a\} = a$$

and affirmation (a) holds for t. Furthermore, x is not discarded by $\mathit{Pop}_R(B_t)$, thus (3) $x \geq B_t$. Affirmation (b) follows using the inductive hypothesis. Now, let I'_k be the interval given by affirmation (c) for $t+1$ and P_k. Let I''_k be the interval of P_k delimited by t and $t+1$. Defining $I_k = I'_k \cup I''_k$, we have that I_k is an interval of P_k, since both I'_k and I''_k contain $t+1$. Also, I_k is delimited as required, as I'_k and I''_k do. Moreover, using the hypothesis $b_t \leq m_t \leq m_t^k$ and $B_t \geq M_t \geq M_t^k$, properties (1)-(3) above and the inductive hypothesis on I'_k we obtain:

$$\mathit{min}(I_k) = \min\{\mathit{min}(I'_k), \mathit{min}(I''_k)\} \geq \min\{a', m_t^k\} \geq \min\{a, m_t\} \geq \min\{a, b_t\} = a$$
$$\mathit{max}(I_k) = \max\{\mathit{max}(I'_k), \mathit{max}(I''_k)\} = \max\{x, M_t^k\} = x.$$

(ii) there exists $a'' > a$ such that $x \in \mathit{Set}_R^{t+1}(a'')$. Then a'' is necessarily discarded by $\mathit{Pop}_L(b_t)$, thus $a'' > b_t$ and $\mathit{Set}_R^{t+1}(a'')$ goes entirely into $\mathit{Set}_R^t(b_t)$. Then, we deduce $x \in \mathit{Set}_R^t(b_t)$ and therefore $x \in \mathit{Set}_R^t(a) \cap \mathit{Set}_R^t(b_t)$, which is impossible unless (4) $a = b_t$. Now, by the inductive hypothesis, $x \in \mathit{Set}_R^{t+1}(a'')$ implies by affirmation (a) that $a'' = \min\{b_w \,|\, t+1 \leq w \leq x-1\} = a'$. Thus, recalling that $a < a''$ by the hypothesis of case (ii), we have (5) $a' = a'' > a$. With (4) and (5) we deduce:

$$\min\{b_w \,|\, t \leq w \leq x-1\} = \min\{b_t, a''\} = \min\{b_t, a'\} = \min\{a, a'\} = a$$

and affirmation (a) holds for t. Furthermore, x is not discarded by $\mathit{Pop}_R(B_t)$, thus (6) $x \geq B_t$. Affirmation (b) follows using the inductive hypothesis. As before, let I'_k be the interval given by affirmation (c) for $t+1$ and P_k. Let I''_k be the interval of P_k delimited by t and $t+1$. Defining $I_k = I'_k \cup I''_k$, we have again that I_k is an interval of P_k delimited as required. Moreover, using the hypothesis $b_t \leq m_t \leq m_t^k$ and $B_t \geq M_t \geq M_t^k$, properties (4)-(6) above and the inductive hypothesis on I'_k:

$$\mathit{min}(I_k) = \min\{\mathit{min}(I'_k), \mathit{min}(I''_k)\} \geq \min\{a'', m_t^k\} \geq \min\{a, m_t\} \geq \min\{a, b_t\} = a$$
$$\mathit{max}(I_k) = \max\{\mathit{max}(I'_k), \mathit{max}(I''_k)\} = \max\{x, M_t^k\} = x.$$

(iii) there is no a'' such that $x \in \mathit{Set}_R^{t+1}(a'')$. In this case, x is added to R during **for**$_t$, in step 8. Then $x = t+1 = B_t$, $a = b_t$ and affirmations (a) and (b) clearly hold. Recall that, by definition, $t+1 \leq M_t^k \leq M_t \leq B_t$ thus $M_t^k = t+1 = x$. Now, the interval I_k delimited by t and $t+1$ on P_k satisfies:

$$\mathit{min}(I_k) \geq m_t \geq b_t = a$$
$$\mathit{max}(I_k) = M_t^k = t+1 = x$$

The "⇒" part of the claim is proved.

Proof of "⇐:" Let a, x and I_k, $2 \leq k \leq K$, be defined as in affirmations (a)-(c) in the claim. Consider the two following cases:

(i) $t < x-1$. Notice that a', x and the intervals I'_k, $2 \leq k \leq K$, delimited on P_k by the leftmost and rightmost values between $t+1, \ldots, x$ satisfy affirmations (a)-(c) for $t+1$, thus by the inductive hypothesis $x \in \mathit{Set}_R^{t+1}(a')$. Now, we show that in both cases occurring during **for**$_t$, we have (7) $\mathit{Set}_R^{t+1}(a') \subseteq \mathit{Set}_R^t(a)$. Indeed, when a' is not discarded by $\mathit{Pop}_L(b_t)$, we necessarily have $a' \leq b_t$ and thus we also have by affirmation (a) that $a = \min\{b_t, a'\} = a'$. Property (7) follows. When a' is discarded, we necessarily have $b_t < a'$ and thus, by affirmation (a), we deduce $a = \min\{b_t, a'\} = b_t$. The execution of $\mathit{Pop}_L(b_t)$ implies, in this case, that $\mathit{Set}_R^{t+1}(a')$ becomes a part of $\mathit{Set}_R^t(a)$, and (7) is proved. Now, with (7) and given that, in step 6, $x = B_{x-1} \geq B_t$ implies that x is not discarded by $\mathit{Pop}_R(B_t)$, we obtain that $x \in \mathit{Set}_R^t(a)$.

(ii) $t = x-1$. Then by hypothesis $a = b_{x-1}, x = B_{x-1}, \mathit{min}(I_k) \geq b_{x-1}$ and $\mathit{max}(I_k) = x$. In step 5 of **for**$_{x-1}$, the instruction $\mathit{Pop}_L(b_{x-1})$ discards all $a'' > b_{x-1}$ (if any). Thus, at the end of step 5, $\mathit{top}(L) \leq b_{x-1}$. The instruction $\mathit{Pop}_R(x)$ insures that, at the end of step 6, $\mathit{top}(R) > x$. (Notice that $x = t+1$ and all the elements pushed before on R by Push_{LR} are of the form $t'+1$ with $t' > t$). Then, the instruction $\mathit{Push}_{LR}(b_{x-1}, x)$ pushes b_{x-1} on L if necessary, pushes x on R (thus $\mathit{top}(L) = b_{x-1}$ and $\mathit{top}(R) = x$) and adds x to $\mathit{Set}_R^t(b_{x-1})$.

Claim 1 is now proved. ∎

Proof of Theorem 1. By definition, $(t..x) \in \mathit{Pairs}_t$ iff $x \in \mathit{Set}_R^t(t)$. According to Claim 1, this holds iff affirmations (a)-(c) hold with $a = t$, that is, the following affirmations hold simultaneously:

(a') $t = \min\{b_w \mid t \leq w \leq x-1\}$

(b') $x = B_{x-1} = \max\{B_w \mid t \leq w \leq x-1\}$

(c') for each k, $2 \leq k \leq K$, the interval I_k of P delimited by the leftmost and rightmost values between $t, t+1, \ldots, x$ satisfies $\mathit{min}(I_k) \geq t$ and $\mathit{max}(I_k) = x$.

We show that I_k has the same elements as $(t..x)$ and that $b_t = t$ ($B_{x-1} = x$ is directly given by affirmation (b')). By affirmation (c'), every element in $(t..x)$ is an element of I_k. Conversely, assume by contradiction that u is any element of I_k distinct from $t, t+1, \ldots, x$. Then u is not a delimiter of I_k. Consequently, let t', $t \leq t' \leq x-1$, such that u is between t' and $t'+1$ on P_k. Then $b_{t'} \leq m_{t'} \leq u \leq M_{t'} \leq B_{t'}$. By affirmations (a') and (b'), $t \leq b_{t'}$ and $B_{t'} \leq B_{x-1} = x$. Thus $t \leq u \leq x$, a contradiction.

To show that $b_t = t$, notice that $b_t \leq t$, by the definition of b_t, and $t \leq b_t$ by affirmation (a'). ∎

4 Complexity issues

We separately discuss the implementation of an *LR*-stack, and the running time of the *LR*-Search algorithm.

4.1 The *LR*-stack

The efficient implementation of an LR-stack depends on the need (or not) to implement Find_L. If Find_L is not needed, then L and R may be implemented as lists. Consequently, Pop_L and Pop_R are easily implemented in linear time with respect to the number of elements removed respectively from L and R, whereas Push_{LR} takes constant time. Also, $\mathit{top}(R)$, $\mathit{top}(L)$ and $\mathit{next}()$ need $O(1)$ time.

When Find_L is needed, then we are in the context of a Union-Find-Delete structure, where the operations are performed on the sets $\mathit{Set}_R(a)$, as follows: unions are performed by Pop_L and Push_{LR}, whereas deletions are performed by Pop_R. These algorithms are already very efficient in the most general case [1], but unfortunately not linear. Yet, particular linear cases may be found and show useful (see Algorithm 2).

4.2 The *LR*-algorithm

We prove the following result.

Theorem 2. *Assume that computing b_t and B_t takes negligible time and space with respect to the computation of m_t^k, m_t, M_t^k and M_t in steps 1-2. Then the LR-Search algorithm has running time $O(Kn+F)$ and uses $O(n+f)$ additional space, where F and f respectively denote the running time and additional space needed by the Filter procedure, over all values of t.*

Proof of Theorem 2. Note that the $Find_L$ operation on the LR-stack is not needed in the algorithm. Therefore, the LR-stack may be easily implemented so as to ensure linear running times for $Pop_L(b_t)$ and $Pop_R(B_t)$ with respect to the number dl_t and dr_t of discarded elements, and constant running time for $Push_{LR}$. Then the **for** loop in steps 4-11 takes running time $O(\Sigma_{1\leq t<n}(dl_t + dr_t + F_t))$, where F_t is the running time of the Filter procedure for t, that is $O(\Sigma_{1\leq t<n}(dl_t + dr_t) + F)$. Furthermore, each variable b_t and B_t is pushed on the *LR*-stack at most once (step 8), and thus it is discarded from the *LR*-stack at most once. Consequently, $\Sigma_{1\leq t<n}(dl_t + dr_t)$ is in $O(n)$ and the **for** loop takes $O(n + F)$ total time. Concerning the memory requirements, it is clear that L and R are filled in with elements b_t, B_t whose cardinality is in $O(n)$.

Given the hypothesis that computing the pairs $[b_t, B_t]$ for $t \in [n-1]$ takes negligible time and space, it remains to show that the other computations in steps 1 and 2 take $O(Kn)$ time and $O(n)$ additional space. To this end, we compute m_t^k and M_t^k, $t \in [n-1]$, for each permutation P_k in $O(n)$ time and $O(n)$ additional space, as described below. The values computed for each permutation are progressively included in the computation of m_t, M_t, so as to use a global $O(n)$ space.

In [7, 4], authors solve a problem called range minimum query (abbreviation: RMQ problem). More precisely, they show that, given any array A of n numbers, it is possible to preprocess it in $O(n)$ time so as to answer in $O(1)$ any query asking for (the position of) the minimum value between two given positions q_1 and q_2 in A. This result, closely related to computing the least common ancestor of two given nodes in a rooted tree, allows us to compute m_t^k, M_t^k for all t in linear time, for each permutation P_k. Then we are already done.

However, we propose here another algorithm, answering a set Q of queries in $O(n+|Q|)$ time. This algorithm is obviously less powerful than the preceding ones, but has at least two advantages. First, it is conceptually and algorithmically simpler, allowing the reader to immediately simulate executions. Second, it gives another application of *LR*-stacks, needing this time the implementation of the $Find_L$ operation.

Algorithm ComputeInf is given in Algorithm 2. The input is an arbitrary permutation P, signed or not, for which we use the same notations as for P_k but without the subscript k. It is quite easy to notice that the algorithm works similarly for an array.

For a pair (q_1, q_2) with $1 \leq q_1 < q_2 \leq n$, denote

$$Inf(q_1, q_2) = \min\{p_h \mid q_1 \leq h \leq q_2\}.$$

Algorithm ComputeInf computes $Inf(q_1, q_2)$ for all pairs in some given set Q, in $O(n + |Q|)$ time. To see this, we first need the following result.

Claim 2. *Let $h \in [n]$. After the execution of the* **for** *loop in step 3 for h, the LR-stack satisfies the property $i \in Set_R(a)$ iff $Inf(i, h) = a$.*

Proof of Claim 2. Notice that only allowed operations are performed on the *LR*-stack, except that $Push_{LR}(p_h, h)$ is not preceded by $Pop_R(h)$. This operation would have no effect, since at the beginning of the h-th step all the elements in R are already less than h.

Remark that in a L^-R^--stack, a' is L-blocking for a iff $a' > a$, and b' is R-blocking for b iff $b' > b$. We use induction. For $h = 0$, at the beginning of the execution the *LR*-stack is empty, and only $Push_{LR}(n + 1, 0)$ is executed, insuring the claim is true. Assuming the claim true for h, let us show it is true for $h + 1$.

Algorithm 2 The ComputeInf algorithm

Input: Permutation P over $[n]$, $Q \subseteq \{(q_1, q_2) \mid 1 \leq q_1 < q_2 \leq n\}$
Output: Values $\mathit{Inf}(q_1, q_2)$ for all $(q_1, q_2) \in Q$

1: Initialize an empty L^-R^--stack
2: $p_0 \leftarrow n + 1$
3: **for** h from 0 to n **do**
4: $\quad \mathit{Pop}_L(p_h)$ //update $\mathit{Inf}(i, h)$ for all $i < h$ s.t. $\mathit{Inf}(i, h-1) > \mathit{Inf}(i, h)$
5: $\quad \mathit{Push}_{LR}(p_h, h)$ // $\mathit{Inf}(h, h) = p_h$
6: $\quad$ **for** all pairs $(q_1, q_2) \in Q$ such that $h = q_2$ **do**
7: $\quad\quad \mathit{query}(q_1, q_2) \leftarrow \mathit{Find}_L(q_1)$
8: $\quad$ **end for**
9: **end for**

Proof of "⇒". Let $i \in \mathit{Set}_R(a)$ at the end of the execution of the **for** loop for $h + 1$. Two cases may appear:

(i) When $a < p_{h+1}$, we deduce that $\mathit{Pop}_L(p_{h+1})$ did not discard a, thus $i \in \mathit{Set}_R(a)$ at the end of the preceding execution of the **for** loop. By the inductive hypothesis, we deduce that $\mathit{Inf}(i, h) = a$ and thus, with $a < p_{h+1}$, we have that $\mathit{Inf}(i, h + 1) = a$.

(ii) When $a = p_{h+1}$, either i has been added by $\mathit{Push}_{LR}(p_{h+1}, h + 1)$, or some $a' > p_{h+1}$ existed on L before $\mathit{Pop}_L(p_h)$ such that $i \in \mathit{Set}_R(a')$ at the end of the execution of the **for** loop for h. In the first case, $i = h + 1$ and the conclusion obviously holds. In the second case, the inductive hypothesis implies that $\mathit{Inf}(i, h) = a'$. Consequently, we have that $\mathit{Inf}(i, h + 1) = p_{h+1} = a$, since $a' > p_{h+1}$.

Proof of "⇐". By the hypothesis, we assume $\mathit{Inf}(i, h + 1) = a$ at the end of the execution of the **for** loop for $h + 1$.

(i) When $a < p_{h+1}$, we deduce $\mathit{Inf}(i, h) = a$ and by the inductive hypothesis we obtain $i \in \mathit{Set}_R(a)$ at the end of the execution of the **for** loop for h. Since $a < p_{h+1}$, the instruction $\mathit{Pop}_L(p_{h+1})$ does not discard a and the conclusion follows.

(ii) When $a = p_{h+1}$, we deduce $\mathit{Inf}(i, h) > p_{h+1}$, since p_{h+1} occurs only once on P. By the inductive hypothesis, that means $i \in \mathit{Set}_R(a')$ at the end of the execution of the **for** loop for h, with $a' = \mathit{Inf}(i, h) > p_{h+1}$. Consequently, $\mathit{Pop}_L(p_{h+1})$ discards a' and moves i into $\mathit{Set}_R(p_{h+1})$ (which is $\mathit{Set}_R(a)$).

The claim is proved.∎

Claim 3. *For each $(q_1, q_2) \in Q$, the value $\mathit{query}(q_1, q_2)$ returned by Algorithm ComputeInf is equal to $\mathit{Inf}(q_1, q_2)$. Moreover, the algorithm may be implemented to have $O(n + |Q|)$ running time.*

Proof of Claim 3. The value $\mathit{query}(q_1, q_2)$ is computed in step 7 of the **for** loop, when $h = q_2$. By Claim 2, $\mathit{Find}_L(q_1) = \mathit{Inf}(q_1, q_2)$ and we are done.

Concerning the running time, the difficulty comes from the need to implement the operation Find_L. However, we benefit here from a very particular case of the Union-Find-Delete context, where no deletion is performed and the union operations (due to Pop_L and Push_{LR}) always join sets of consecutive elements to obtain another set of consecutive elements. In [16], an implementation of the Union-Find operations for this particular case is proposed, which realizes each union between two sets in $O(1)$, and m operations $Find$ in $O(n + m)$. Moreover, the sets (equivalently: the elements of L) as well as their elements (equivalently: the elements in R) may be easily chained to simulate the L and R stacks. Then each Pop_L operation takes linear time with respect to the number of discarded elements and is in $O(n)$

Algorithm 3 The Filter algorithm for common intervals

Input: Pointers $R^{\top}(t), R^{\perp}(t)$ to the first and last element of $Set_R(t)$ (possibly equal to nil)
Output: All common intervals $(t..x)$ of $\mathcal{P}$, with fixed t.

```
1: if R^⊤(t) ≠ nil then
2:    x^⊤ ← the target of R^⊤(t); x^⊥ ← the target of R^⊥(t)
3:    x ← x^⊤
4:    while x ≤ x^⊥ do
5:       Output the interval (t..x)
6:       x ← next(x)                    //or n + 1 if next(x) does not exist
7:    end while
8: end if
```

time over all h, since the elements on L are elements of P, which are pushed exactly once on L. Furthermore, $Push_{LR}$ takes constant time and the $|Q|$ calls of $Find_L$ take $O(n + |Q|)$ time. The indicated running time for our algorithm follows. ∎

Proof of Theorem 2 (continued). Algorithm ComputeSup, which computes the maximum values between given pairs of positions in P, is clearly similar to ComputeInf. Then, m_t^k and M_t^k may be obtained by appropriately defining the query set Q, and the running time of *LR*-Search follows.∎

For each class $\mathcal{C}$ among common, nested and conserved intervals, the following sections give the settings for b, B and Filter in the *LR*-Search algorithm, prove the correction of the resulting variant of *LR*-Search and discuss subclasses $\mathcal{C}' \subseteq \mathcal{C}$. For the ease of presentation, we assume that $R^{\perp}(a)$ is a pointer to the last element element of $Set_R(a)$, for all $a \in SL$ with $Set_R(a) \neq \emptyset$. It is easy to check that such a pointer is easily updated during the operations on the *LR*-stack.

Remark 4. According to the preceding definitions, the set $Set_R(t)$ is, at the end of step 9 of **for**$_t$, the set denoted $Set_R^t(t)$.

5 Finding common, nested and conserved intervals

5.1 Common intervals

In this case, $\mathcal{P}$ is a set of permutations (all elements have a + sign). The *MinMax*-profile of $\mathcal{P}$ uses in this case the basic settings for b and B:

- $b(i) := m_i$, for all i, $1 \leq i \leq n-1$
- $B(i) := M_i$, for all i, $1 \leq i \leq n-1$
- Filter is given in Algorithm 3.

Example 5. Recall that, in Figure 1, $\mathcal{P} = \{P_1, P_2\}$ with $P_1 = Id_7$ and $P_2 = (7\,2\,1\,3\,6\,4\,5)$. The Filter procedure in Algorithm 3 successively outputs the intervals $(4..5), (4..6)$ (when $t = 4$), $(3..6)$ (when $t = 3$) and $(1..2), (1..3), (1..6), (1..7)$ (when $t = 1$).

Theorem 3. *Algorithm LR-Search with settings b, B and Filter above solves Common-ISP for K permutations in $O(Kn + N)$ time and $O(n)$ additional space, where N is the number of common intervals of $\mathcal{P}$.*

Proof of Theorem 3. By Remark 4, Algorithm 3 considers each x in $Set_R^t(t)$. Thus, over all values of t, Filter outputs $\cup_{1 \leq t \leq n-1} Pairs_t$. By Theorem 1, this is exactly the set of common intervals $(t..x)$ of $\mathcal{P}$ with $b_t = t$ and $B_{x-1} = x$. Thus all the intervals output by the algorithm are common intervals

of $\mathcal{P}$. Conversely, let $(t..x)$ be a common interval of $\mathcal{P}$. Then, for each k, $2 \leq k \leq K$, and each w, $t \leq w \leq x-1$, we must have $t \leq m_w^k \leq w < w+1 \leq M_w^k \leq x$. Consequently, $t \leq m_w \leq w < w+1 \leq M_w \leq x$ for all w, $t \leq w \leq x-1$. By definition, $b_i = b(i) = m_i$ and $B_i = B(i) = M_i$ for all i, and thus we have $t \leq b_w$ and $B_w \leq x$ for all w. We deduce that

$$b_t \leq t \leq \min\{b_w \,|\, t \leq w \leq x-1\}$$

$$B_{x-1} \geq x \geq \max\{B_w \,|\, t \leq w \leq x-1\}$$

and thus $t = b_t$ and $x = B_{x-1}$. By Theorem 1 we obtain that $(t..x)$ is output by *LR*-Search and the correction of the algorithm is proved.

The **while** loop in steps 4-7 of Filter takes $O(|Set_R^t(t)|)$ time, since $R^\top(t) = top(R)$. Filter has therefore a linear complexity with respect to the size of the output. By Theorem 2 and given that $b(i)$ and $B(i)$ are computed in constant time for each i when m_i and M_i are known, the *LR*-Search algorithm with the abovementioned settings runs in $O(n+N)$ time. The additional space used by Filter is in $O(1)$.∎

5.2 Nested Intervals

In this case too, $\mathcal{P}$ is a set of permutations (all elements have + sign) and the *MinMax*-profile uses the basic settings for *b* and *B*:

- $b(i) := m_i$, for all i, $1 \leq i \leq n-1$
- $B(i) := M_i$, for all i, $1 \leq i \leq n-1$
- Filter is given in Algorithm 4.

Example 6. Again, in Figure 1 the *LR*-Search algorithm is applied to $\mathcal{P} = \{P_1, P_2\}$, where $P_1 = Id_7$ and $P_2 = (7\,2\,1\,3\,6\,4\,5)$. When $t = 4$, the Filter procedure in Algorithm 4 outputs intervals $(4..5), (4..6)$ and sets $W[3] \leftarrow 6$. When $t = 3$, $top(L) = 3$ and $W[3] \neq 0$, so that $(3..6)$ is output and $W[2]$ receives the value 6. Now, when $t = 2$ we have $top(L) \neq 2$ and the **if** instruction in step 4 stops the execution of Filter. Then $W[1]$ remains 0. Thus, when $t = 1$, in step 9 of Filter we have $x^\top = t+1$ and the algorithm outputs $(1..2), (1..3)$ but not $(1..6)$, nor $(1..7)$.

In this subsection, we say that $(x, x+1)$ is a *gap* if exactly one of $x, x+1$ is on R.

Theorem 4. *Algorithm LR-Search with settings b, B and Filter above solves Nested-ISP for K permutations in $O(Kn+N)$ time and $O(n)$ additional space, where N is the number of nested intervals of $\mathcal{P}$.*

By Remark 4, $Set_R(t)$ in Algorithm 3 is the same as $Set_R^t(t)$. Notice that the value of $W[t]$ computed by Filter during its execution for $t+1$ indicates the largest element x in $Set_R^{t+1}(t+1)$ such that $(t+1..x)$ is output by Filter (step 13 in Filter for $t+1$). However, $W[t]$ may be set to 0 in step 7 of the Filter procedure (when executed for t), if we know that its initial value was discarded by $Pop_R(B_t)$ in the *LR*-Search algorithm.

From now on, we focus on the execution of the **for** loop in *LR*-Search for t (including Filter). Let y_t be defined as follows. We use the notations in step 5 of the Filter procedure, and assume thus that $R^\top(t) \neq nil$. If $x^\top = t+1$ or $W[t] \neq 0$, then y_t is the first element in $Set_R^t(t)$ such that the following properties hold:

(1) $W[t] \leq y_t$, and

(2) either (y_t, y_t+1) is a gap, or $y_t = x^\perp$.

Otherwise, *i.e.* if $x^\top \neq t+1$ and $W[t] = 0$, y_t is set to 0. Let

Algorithm 4 The Filter algorithm for nested intervals

Input: Pointers $R^{\top}(t), R^{\perp}(t)$ to the first and last element of $Set_R(t)$ (possibly equal to nil)
Output: All nested intervals $(t..x)$ of $\mathcal{P}$ with fixed t.

1: **if** $t = n - 1$ **then**
2: Let W be a n-size vector filled in with 0
3: **end if**
4: **if** $R^{\top}(t) \neq nil$ **then**
5: $x^{\top} \leftarrow$ the target of $R^{\top}(t)$; $x^{\perp} \leftarrow$ the target of $R^{\perp}(t)$
6: **if** $x^{\top} > W[t]$ **then**
7: $W[t] \leftarrow 0$ *// element $W[t]$ has been discarded by $Pop_R(B_t)$*
8: **end if**
9: **if** $x^{\top} = t + 1$ or $W[t] \neq 0$ **then**
10: $x \leftarrow x^{\top}$
11: **while** $x \leq x^{\perp}$ and ($x = t + 1$ or $x \leq W[t]$ or $(x - 1, x)$ is not a gap) **do**
12: Output the interval $(t..x)$
13: $W[t-1] \leftarrow x$ *// t passes down to $t-1$ its largest x such that $(t..x)$ is output*
14: $x \leftarrow next(x)$ *//or $n+1$ if $next(x)$ does not exist*
15: **end while**
16: **end if**
17: **end if**

$$V_t := \{v \in Set_R^t(t) \mid x^{\top} \leq v \leq y_t\}.$$

Now, let us prove that:

Claim 4. *The intervals output by Filter are the intervals $(t..x)$ with $x \in V_t$.*

Proof of Claim 4. Indeed, when $x^{\top} \neq t + 1$ and $W[t] = 0$, the condition in step 9 is false and the algorithm returns an empty set of intervals. This is correct, since $V_t = \emptyset$ in this case.

In the contrary case, the condition in step 9 is true. The **while** loop starts with $x = x^{\top}$ and outputs all intervals $(t..x)$ with x in $Set_R^t(t)$ up to a last one $(t..x_0)$. If $x_0 = x^{\perp}$, then $x_0 = x^{\perp} = y_t$, since $x^{\perp}$ satisfies properties (1) and (2) in the definition of y_t whereas no other element preceding it in $Set_R^t(t)$ does. If $x_0 \neq x^{\perp}$, then x_0 satisfies the second condition in step 11, whereas the element $x' = next(x)$ is in $Set_R^t(t)$ but does not satisfy the second condition in step 11. Then $(x' - 1, x')$ is a gap and thus $(x_0, x_0 + 1)$ is a gap too. In order to deduce that $x_0 = y_t$, we only have to show that condition (1) in the definition of y_t is satisfied by x_0. If, by contradiction, $W[t] > x_0$, then $W[t]$ is on R below x_0 and thus $x' \leq W[t]$. This is impossible, since x' does not satisfy the second condition in step 11. ∎

The following claim establishes the correction of our algorithm:

Claim 5. *We have $x \in V_t$ iff $(t..x)$ is a nested interval.*

Proof of Claim 5. This proof is by induction on t. For $t = n-1$, $R^{\top}(n-1) \neq nil$ iff $Push_{LR}(b_t, t+1)$ is called in *LR*-Search with $b_t = n - 1$ and $t + 1 = n$. Thus $Set_R^{n-1}(n - 1) = \{n\}$. Then, with $W[n - 1] = 0$, we have that $V_{n-1} = \{n\}$ and indeed $(n - 1..n)$ is a nested interval.

We now assume the claim is true for $t + 1$ and show it for t. Then $R^{\top}(t) \neq nil$ in step 4 of Filter (otherwise V_t is not defined, and there is no nested interval with left endpoint t) and thus $Pop_L(b_t)$ in step 5 of *LR*-Search has been executed with $b_t = t$. Consequently, $V_{t+1} \subseteq Set_R(t)$ at the end of step 5 in $\mathbf{for}_t$ of *LR*-Search. In step 6, all elements x of R with $x < B_t$ are discarded and, with them, the first elements of V_{t+1} (since the first element of $Set_R(t)$, as well as of V_{t+1} if it is non-empty, is $top(R)$). Furthermore, the $Push_{LR}(b_t, t+1)$ operation, if executed, pushes $t+1$ on the top of R and, in the same time, in $Set_R^t(t)$.

When Filter is executed for t, several situations may occur. Notations $x^\top$ and $x^\perp$ concern the execution of the **for** loop in *LR*-Search, including Filter, for t. Then $x^\top = \mathit{top}(R)$.

(i) $x^\top = t+1$ and $W[t] = 0$ in step 9 of Filter. Then either there is no x such that $(t+1..x)$ is output, or such an x exists but is removed from R by $\mathit{Pop}_R(B_t)$. According to the definition, y_t is the first element of $\mathit{Set}_R^t(t)$ such that either (y_t, y_t+1) is a gap, or $y_t = x^\perp$. In both cases, since $x^\top = t+1$ we have that $t+1$ is in V_t. Also, $(t..t+1)$ is nested since it is a common interval (by Theorem 1) and has cardinality 2. By the definition of nested intervals, all intervals $(t..x)$ with successive values of x, $x \geq t+2$, will be nested, thus $x \in V_t$ implies that $(t..x)$ is nested. Conversely, assume by contradiction that some $x \geq t+2$ exists such that $(t..x)$ is nested but $x \notin V_t$, and take x as small as possible with these properties. Since $(t..x)$ is nested, $(t..x)$ is a common interval thus, by Theorems 1 and 3 and given the settings of b and B, we have $x \in \mathit{Set}_R^t(t)$, thus $x \leq x^\perp$. As $(t..x)$ is nested, either $(t+1..x)$ or $(t..x-1)$ is nested. The former does not hold (because of $W[t] = 0$), thus $(t..x-1)$ is nested. Since x was the smallest with the indicated properties, $x-1 \in V_t$. But then, by the definition of y_t, we have $x \in V_t$ too, a contradiction.

(ii) $x^\top = t+1$ and $W[t] > 0$ in step 9 of Filter. Then

$$V_t = \{t+1\} \cup V_{t+1} - \{v \in V_{t+1} \mid v < t+1\} \cup U$$

where U is the set of all consecutive elements $W[t]+1, W[t]+2, \ldots, z \subseteq \mathit{Set}_R^t(t)$ such that either $z = x^\perp$ or $(z, z+1)$ is a gap. Equivalently, V_t gets all the elements in V_{t+1} except those smaller than $t+1$ (because of Pop_R), as well as $t+1$ (pushed by Push_{LR}) and all the consecutive elements that are possibly added at the end of V_{t+1} during the $\mathit{Pop}_L(b_t)$ operation, with $b_t = t$. It is easy to see that the last element of U satisfies the conditions (1) and (2) in the definition of y_t, and no one before it in $\mathit{Set}_R(t)$ does. Thus $z = y_t$.

By contradiction, assume some $x \in V_t$ exists such that $(t..x)$ is not nested. Assume x is the smallest with these properties. Then $x > W[t]$, otherwise $x = t+1$ or $x \in V_{t+1}$ and then $(t+1..x)$ is nested by definition and the inductive hypothesis, insuring that $(t..x)$ is nested. Now, $x-1 \in V_t$ (since there is no gap in V_t beyond $W[t]$) thus, by the minimality of x, $(t..x-1)$ is nested. But then $(t..x)$ is nested, a contradiction. Conversely, let $(t..x)$ be a nested interval, and let us show that $x \in V_t$. Once again, assume this is not true and let x be the smallest counter-example. Since $(t..x)$ is nested, $(t..x)$ is a common interval and, by Theorems 1 and 3, $x \in \mathit{Set}_R^t(t)$ thus $x \geq t+1$. Now, we must have $x > W[t]$, otherwise $x = t+1$ or $x \in V_{t+1}$ thus $x \in V_t$, a contradiction. Finally, since $(t..x)$ is nested we have two cases. Either $(t+1, x)$ is nested, and then by the inductive hypothesis $x \in V_{t+1}$ thus $x \in V_t$, a contradiction. Or $(t, x-1)$ is nested and thus $x-1 \in V_t$ by the minimality of t, thus $x \in V_t$ by the definition of V_t, another contradiction.

(iii) $x^\top \neq t+1$ and $W[t] > 0$ in step 9 of Filter. Then

$$V_t = V_{t+1} - \{v \in V_{t+1} \mid v < t+1\} \cup U$$

with U defined as previously done, and the proof follows similarly.

(iv) $x^\top \neq t+1$ and $W[t] = 0$ in step 9 of Filter. Then $V_t = \emptyset$ and we must show there is no nested interval $(t..x)$. If, by contradiction, such an x exists, then assume x is taken to be the smallest one. Then $(t..x-1)$ is not nested, by the minimality of x. Thus $(t+1, x)$ is nested, and thus $x \in V_{t+1}$. But then $x \in V_t$ unless x is removed by $\mathit{Pop}_R(B_t)$, with $B_t = t+1$. However, this is impossible, since $x \geq t+1$.∎

Remark 5. According to the preceding claim, $W[t] = y_{t+1}$. Moreover, assume that every element r in R has an associated pointer $R^g(r)$ on the first element $w \in R$ larger than or equal to r and such that

either $(w, w+1)$ is a gap or w is the bottom of R. Then y_t may be computed in constant time in each of the cases (i)-(iv) of the proof, using $W[t]$ (*i.e.* y_{t+1}) and $R^g(r_0)$, where r_0 is the target of $R^\top(t)$ at the end of **for**$_{t+1}$, if $R^\top(t) \neq nil$.

Proof of Theorem 4. The algorithm correction is proved by Claims 4 and 5. The $O(Kn+N)$ running time of the algorithm is due to Theorem 2 and to the linearity of Filter with respect to $|V_t|$, which is clear for $t < n-1$ since Filter stops when the first element not in V_t is found. The $O(n)$ complexity when $t = n-1$ does not change the overall running time of the algorithm.∎

5.3 Conserved intervals

Here, each P_k is a signed permutation with $p_1^k = 1$ and $p_n^k = n$, both positive. The definitions of b and B define a *MinMax*-profile of $\mathcal{P}$ adapted to the specific needs of conserved intervals. For each i with $1 \leq i \leq n-1$, let $u_i^k := m_i^k$ if $m_i^k = i$, and $u_i^k := m_i^k - 1$ otherwise. Similarly, let $v_i^k := M_i^k$ if $M_i^k = i+1$, and $v_i^k := M_i^k + 1$ otherwise. Let:

- $b(i) := \min\{u_i^k \mid 2 \leq k \leq K\}$, for all i, $1 \leq i \leq n-1$
- $B(i) := \max\{v_i^k \mid 2 \leq k \leq K\}$, for all i, $1 \leq i \leq n-1$
- Filter is given in Algorithm 5, where: $R^\top(t)^*$ is a pointer to the first element x in $Set_R(t)$ such that t and x have the same sign in all permutations in $\mathcal{P}$ (such elements x are chained together inside R); and *Position*$(t, t+1)$ returns true iff, for each k, either t is positive in P_k and $P_k^{-1}(t) < P_k^{-1}(t+1)$, or t is negative in P_k and $P_k^{-1}(t) > P_k^{-1}(t+1)$.

Remark 6. Note that $(t..x)$ is a conserved interval of $\mathcal{P}$ iff it has the following properties:
(1) it is a common interval of $\mathcal{P}$
(2) it is delimited by t and x on P_k, for all $k \in [K]$
(3) t and x are both positive or both negative in each P_k, for all $k \in [K]$
(4) for each $k \in [K]$, either t is positive in P_k and $P_k^{-1}(t) < P_k^{-1}(x)$, or t is negative in P_k and $P_k^{-1}(x) < P_k^{-1}(t)$.

Conditions (1) and (2) are easily handled by defining the bounding functions b and B as indicated. However, conditions (3) and (4) need a preprocessing of the permutations in $\mathcal{P}$, in order to: (Task 1) identify and chain together inside R the elements x in the same equivalence class with respect to the relation "x and x' have the same sign in all permutations", allowing us to deal with (3); and (Task 2) compute the boolean function *Position*() defined above, which will insure that (4) holds. These tasks are done in $O(Kn)$ time and $O(n)$ additional space as follows:

Task 1. Consider the matrix M whose row vectors are the vectors $sign_{P_k}$ for $k \in \{2, \ldots, K\}$, and perform a radix sort on the columns of this matrix (which correspond to the elements t of the permutations). Group together all elements t that have the same column vector, i.e. the same sign in all permutations. For each group s, the elements of the group that are pushed on R are progressively chained together immediately after $Push_{LR}(b_t, t+1)$ (step 8 in *LR*-Search), and the pointer *First*(s) to the first element of the chain is updated as needed. Then, for each t, $R^\top(t)^*$ is defined as *First*(s), where s is the group of t. All these operations are done in $O(Kn)$ time and $O(n)$ additional space, assuming that radix sort does not really create the indicated matrix, but rather manipulates column numbers and checks the values directly on the vectors $sign_{P_k}$ (which are not modified).

Task 2. The relative positions of t and $t+1$ in each P_k are checked in $O(1)$ using the functions $P_k^{-1}()$, and thus *Position*$(t, t+1)$ is computed in $O(K)$ time and $O(1)$ additional space, for each $t \in [n]$.

With this supplementary information, the Filter procedure is presented in Algorithm 5.

Algorithm 5 The Filter algorithm for conserved intervals

Input: Pointers $R^{\top}(t)^*$ to the first element in $Set_R(t)$ having the same sign as t in all permutations P_k, and $R^{\perp}(t)$ to the last element in $Set_R(t)$ (they are possibly nil)
Output: All conserved intervals $(t..x)$ of $\mathcal{P}$ with fixed t.

```
1: if R^⊤(t)* ≠ nil then
2:    x^⊤ ← the target of R^⊤(t)*; x^⊥ ← the target of R^⊥(t)
3:    x ← x^⊤
4:    if Position(t, t+1) then
5:       while x ≤ x^⊥ do
6:          Output the interval (t..x)
7:          x ← the element immediately following x in its chain      //or n+1 if it does not exist
8:       end while
9:    end if
10: end if
```

Example 7. In Figure 2, conserved intervals are computed for $\mathcal{P} = \{P_1, P_2\}$, where $P_1 = Id_7$ and $P_2 = (1\,3\,2\,6\,4\,5\,7)$ with positive $1, 6$ and 7, and negative $2, 3, 4, 5$. Note, for, instance, that $m_6 = 4$ but $b_6 = 3$, by the definition of b_t, indicating that 4 cannot be the delimiter of a conserved interval containing 6 and 7. Similar remarks are valid for b_5, B_3, b_3 and B_1. During the execution of *LR*-Search, Filter does not output $(4..5)$ since $Position(4, 5) = false$, but outputs $(2..3)$ and $(1..7)$ for which variable $Position()$ is true and the signs are compatible. These are the conserved intervals of $\mathcal{P}$.

Theorem 5. *Algorithm LR-Search with settings b, B and Filter above solves Conserved-ISP for K signed permutations in $O(Kn + N)$ time and $O(n)$ additional space, where N is the number of conserved intervals of $\mathcal{P}$.*

We start by showing that:

Claim 6. *An interval $(t..x)$ with the properties* (1), (2) *in Remark 6 also satisfies property* (4) *iff $Position(t, t + 1)$ is true.*

Proof of Claim 6. The "⇒" part is obviously true. For the "⇐" part, assume by contradiction and without any loss of generality that k and x, with $x > t$, exist such that t is positive and $P_k^{-1}(x) < P_k^{-1}(t)$. Then, the hypothesis that $Position(t, t + 1)$ is true insures that $P_k^{-1}(t) < P_k^{-1}(t + 1)$, thus $t + 1$ does not belong to the interval delimited by t and x. Consequently, condition (1) in Remark 6 is contradicted. The reasoning is similar if t is negative.∎

Proof of Theorem 5. We show that $(t..x)$ is output by *LR*-Search with the given parameters iff it is a conserved interval. Recall that, by Remark 4, $Set_R(t)$ in Algorithm 3 is the same as $Set_R^t(t)$. We assume without loss of generality that t is positive.

Proof of "⇒:" According to Theorem 1, the set of intervals computed by the algorithm *LR*-Search is the set of common intervals (ignoring the signs) $(t..x)$ of $\mathcal{P}$ with $b_t = t$ and $B_{x-1} = x$. Then, for each output $(t..x)$ and each k, the corresponding interval on P_k has property (1) in Remark 6. To check property (2), notice that by Theorem 1 we have $t = b_t$ and $x = B_{x-1}$, thus according to the conditions on $b()$ and $B()$:

$$m_t \leq t = b_t = b(t) \leq m_t \text{ and } M_{x-1} \geq x = B_{x-1} = B(x-1) \geq M_{x-1}$$

We deduce that $b_t = m_t = t$ and $B_{x-1} = M_{x-1} = x$. By the second part of Theorem 1, we know that $b_t \leq b_w$ and $B_{x-1} \geq B_w$ for all w, $t \leq w \leq x - 1$. Assume by contradiction that t is not a delimiter of the interval of P_k made of $t, t + 1, \ldots, x$. Then, there is some w, $t + 1 \leq w \leq x - 1$ such that t is between w and $w + 1$ on P_k. Then $m_w \leq m_w^k = t < w$ and thus $b_w = b(w) \leq m_w - 1 \leq t - 1$. But then $b_t = t > t - 1 \geq b_w$, a contradiction. The reasoning is similar for x. Property (2) is proved.

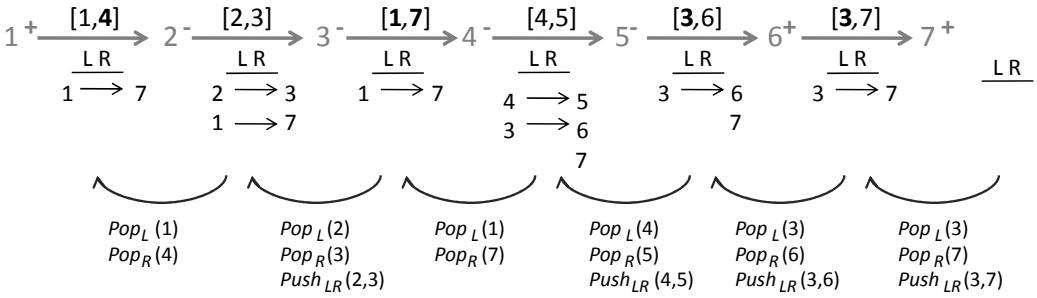

Figure 2: The *LR*-Search algorithm for conserved intervals when $P_2 = (1\,3\,2\,6\;4\,5\,7)$ with the signs indicated on the figure. The chains in R may be easily deduced using the signs. Bounds b_t and B_t are in bold font whenever they are not equal to m_t and respectively M_t.

Property (3) is insured by the interpretation of $R^{\top}(t)^*$ and $R^{\perp}(t)$, as well as by steps 1-3 and 7 in Filter. Claim 6 and step 4 in Filter guarantee that the property (4) holds. Thus $(t..x)$ is a conserved interval.

Proof of "$\Leftarrow$:" We have $b_t = b(t) = m_t = t$ and $B_{x-1} = B(x-1) = M_{x-1} = x$, by the definition of b, B and since a conserved interval is a common interval. By Theorem 1, we deduce that $x \in Set_R^t(t)$. Furthermore, we use properties (1)-(4) in Remark 6 and show that no interval with these properties is forgot by Filter. By contradiction, if this was the case, then x would be eliminated by the condition in step 4 of Filter. But then, by Claim 6 the interval $(t..x)$ cannot satisfy property (4) in Remark 6. This contradicts the assumption that $(t..x)$ is conserved.

It is easy to see that b_t and B_t may be computed in $O(Kn)$ time and $O(n)$ additional space using the values m_t^k, M_t^k (and avoiding to store all these values), thus we may apply Theorem 2. Filter clearly has running time proportional to the number of output intervals, since the **while** loop in step 5 is executed only when the condition in step 4 is true. Moreover, the **while** loop has running time proportional to the number of output intervals. Theorem 2 finishes the proof.∎

6 Finding subclasses of common, nested and conserved intervals

6.1 Irreducible common intervals, and other common intervals

Irreducible common intervals have been defined in [12] as follows. Let G be the graph whose vertices are all the common intervals of $\mathcal{P}$, and whose edges are the pairs of non-disjoint common intervals. Then a common interval I is *reducible* if it is the set union of some of its proper sub-intervals that are common intervals and induce together a connected subgraph of G. Otherwise, I is *irreducible*.

Consider now the total order on the set of common intervals of $\mathcal{P}$ given by $(t_1..x_1) < (t_2..x_2)$ iff either $t_1 > t_2$, or $t_1 = t_2$ and $x_1 < x_2$. For each w with $1 \leq w \leq n-1$, let $Small\,(w)$ denote the smallest, with respect to this order, common interval of $\mathcal{P}$ containing w and $w+1$. It is shown in [12] that:

Claim 7. *The set of irreducible intervals of $\mathcal{P}$ is the set $\{Small\,(w) \mid 1 \leq w \leq n-1\}$.*

Then [12] proposes an algorithm to solve IrreducibleCommon-ISP in linear time. Another linear time algorithm is obtained by appropriately filtering the results of our *LR*-Search algorithm, as shown below.

Claim 8. *Let w be an integer such that $1 \leq w \leq n-1$. The interval Small (w) is the common interval $(t..x)$ of $\mathcal{P}$ such that the couple $(n-t, x)$ of integers is minimum with respect to the lexicographic order with the property $t \leq w < x$.*

Proof of Claim 8. The condition involving the lexicographic order means that we first maximize t, and then minimize x. By contradiction, assume that the interval $(t..x)$ is not *Small* (w). Now, *Small* (w) is minimum according to the total order. We must then have either $t <$ *min*(*Small* (w)) (the smallest element in *Small* (w)), or $t =$ *min*(*Small* (w)) and $x >$ *max*(*Small* (w)). But this contradicts the choice of t and x.∎

The *LR*-Search algorithm we propose here uses the same *MinMax*-profile for $\mathcal{P}$ as common intervals, that is $b(i) := m_i$ and $B(i) := M_i$. However, the Filter algorithm is in this case a refinement of the initial Filter procedure in Algorithm 3. Algorithm 6 uses a stack S to store the values w for which *Small* (w) has not been found yet, in increasing order from top to bottom of S. For a given w, the first interval $(t..x)$ such that $t \leq w < x$ found by *LR*-Search is *Small* (w). However, such a value x may be located in $Set^t_R(t)$ after a sequence of useless values x'. At each moment of the execution of Filter, say that a value x on R is *untrusty* if at least one interval $(t'..x)$ has been already output (meaning that x has possibly become useless), and *trusty* otherwise. Call a *strip* any maximal sequence of consecutive integers on R that are all untrusty.

In order to insure the best running time for our algorithm, we consider that each element x on R but the bottom of R has, in addition to its successor *next*(x) on R, another successor denoted *nextt*(x):

- if x is trusty, then its successor *nextt*(x) is *next*(x);
- if x is untrusty and is the head (i.e. the first) element in its strip, then *nextt*(x) is the first element following the strip (if such an element exists)
- if x is untrusty and it is not the head element in its strip, then *nextt*$(x) =$ *nextt*(x'), where x' is the head element in the strip of x.

Remark 7. Note that the state (trusty or untrusty) of an element may be easily computed in Filter. Moreover, the successor *nextt*(x) of an element x is computed in $O(1)$ when x is pushed on R by $Push_{LR}$. Furthermore, when an element x becomes untrusty (step 10 in Filter), the successors *nextt*$()$ change in the strip before and possibly in the strip after x on R. We call Update(*nextt*$()$) the procedure performing this update in Algorithm 6 (step 10). We do not give its details here, but discuss it in the proof of Theorem 6.

Example 8. On the example in Figure 1, the *LR*-Search algorithm with the Filter procedure in Algorithm 6 first outputs (when $t = 4$) $(4..5)$ and $(4..6)$ (which are *Small* (4) and *Small* (5)). The values 5 and 6 become untrusty (step 10). When $t = 3$, the interval $(3..6)$ is output, which is *Small* (3). Finally, with $t = 1$, intervals $(1..2)$ and $(1..3)$ are successively output (they are *Small* (1) and *Small* (2)). The stack S still contains 6 and the next value in $Set_R(1)$ is $x = 6$. Thus *top*$(S) = x = 6$ in step 8, and x is untrusty. In step 16, x is updated to the value *nextt*(6) which is 7. The interval $(1..7)$, which is *Small* (6), is output.

Theorem 6. *Algorithm LR-Search with settings b, B as for common intervals and the more restrictive Filter procedure in Algorithm 6 solves IrreducibleCommon-ISP for K permutations in $O(Kn)$ time and $O(n)$ additional space.*

Proof of Theorem 6. The decreasing order of t in the **for** loop of the *LR*-Search algorithm and the increasing values of the elements in $Set^t_R(t)$ show that the intervals $(t..x)$ are considered according to the lexicographic order required by Claim 8. Values w are pushed on S as soon as w is considered (step 4 in Filter) and are discarded from S iff the interval with maximum t and minimum x containing w and $w+1$ is output (steps 10 and 12). By Claim 8, this interval is *Small* (w). Thus, all the intervals output by the algorithm are irreducible.

Algorithm 6 The Filter algorithm for irreducible common intervals

Input: Pointers $R^\top(t), R^\perp(t)$ to the first and last element of $Set_R(t)$ (possibly equal to nil)
Stack S and values *nextt*() output by the preceding Filter call (except when $t = n-1$).
Output: All irreducible common intervals $(t..x)$ of $\mathcal{P}$ with fixed t.

```
1: if t = n − 1 then
2:    Let S be an empty stack
3: end if
4: Push t on S                                          // Small(t) has not been found yet
5: if R^⊤(t) ≠ nil then
6:    x^⊤ ← the target of R^⊤(t); x^⊥ ← the target of R^⊥(t)
7:    x ← x^⊤
8:    while x ≤ x^⊥ and S is not empty and (top(S) < x or x is untrusty) do
9:       if top(S) < x then
10:         Output the interval (t..x); Update(nextt())          //x becomes untrusty
11:         while S is not empty and top(S) < x do
12:            Pop top(S) from S          // for top(S), interval Small(top(S)) has been output
13:         end while
14:         x ← next(x)                                  //or n + 1 if next(x) does not exist
15:      else
16:         x ← nextt(x)                                 //or n + 1 if nextt(x) does not exist
17:      end if
18:   end while
19: end if
```

Conversely, assume by contradiction that some interval *Small* (w) is not output by the algorithm, and let w be the smallest such value. Let *Small* $(w) = (t..x^*)$ and consider the execution of Filter for t. Since $(t..x^*)$ is not output by the algorithm, the execution of the **while** loop in step 8 either (i) misses x^* by skipping it in step 16, or (ii) stops before x^* is reached. Notice that $x^* \in Set^t_R(t)$ (by Theorem 1). Let x' be the largest element smallest than x^* for which the condition in step 8 of Filter is tested, and consider the state of the stack S immediately after this test.

Let us show that $top(S) = w$. By contradiction, if we assume $u := top(S) < w$ then the minimality of w insures that *Small* (u) is output by the algorithm *LR*-Search with the given settings. Then *Small* $(u) = (t_0..x_0)$ with $t_0 \leq t$, since u is still on S. Moreover, $u > t$ since $Push_{LR}$ has pushed on R only elements $t'+1$ with $t' \geq t$. Now, we cannot have $t_0 < t$ since then $t_0 < t < u < w < x^*$ and thus $(t..x^*)$ is smaller than *Small* (u) and contains both u and $u+1$, a contradiction. We thus have $t_0 = t$. Now, $(t..x_0)$ is certainly output by Filter, and this has not been done yet when x' is considered (otherwise, u would have been discarded from S). We deduce that x' does not stop the execution of the **while** loop, thus $u < x'$ or x' is untrusty. The former one would contradict the maximality of x', because of step 14 which brings into step 8 a value larger than x'. The latter one implies that the next trusty value is larger than x^* (since x^* is skipped because of $x' \leftarrow nextt(x')$), and thus $x_0 > x^*$ and $(t..x^*) <$ *Small* (u) contradicts the minimality of *Small* (u). Thus $top(S) = w$.

Consider now the cases (i) and (ii) before.

(i) In this case, x' is the value for which step 16 has been executed. Then x' and x^* are untrusty (by the definition of *nextt*()), $x^* > x'$ (otherwise x^* is not skipped) and the values in $Set^t_R(t)$ between x' and x^* are consecutive and untrusty (they belong to the same strip). Moreover, step 16 has been executed, so that the condition in step 9 of Filter is not verified. Thus $top(S) \geq x'$. Since $top(S) = w$ and $w < x^*$ (recall that $(t..x^*)$ contains w and $w+1$) we deduce that $x^* > w \geq x'$. Then w is between x' and $x^* - 1$ on the strip, and $w+1$ is between $x'+1$ and x^* on the strip. Consecutively, $w+1$ is untrusty, thus an interval $(t'..w+1)$ with $t' > t$ has been already output by Filter. But this interval is smaller that $(t..x^*)$ and should therefore be *Small* (w), a contradiction.

(ii) In this case, x' satisfies $x' < x^*$ and $x' \leq \mathit{top}(S)$ and x' is trusty. Now, $x'-1$ is not on S, since $w = \mathit{top}(S)$ and elements in S are in increasing order from top to bottom. Thus, by the minimality of w, $\mathit{Small}\,(x'-1)$ is output by the algorithm. Let $\mathit{Small}\,(x'-1) = (t''..x'')$ and notice that $x'' > x'$ (x' must be contained in $\mathit{Small}\,(x'-1)$ but x' has not become untrusty) and $t'' > t$ (otherwise, $x'-1$ should be on S). Then $(t..x') \cap (t''..x'') = (t''..x')$ and it is a common interval containing both $x'-1$ and x' and which is smaller than $\mathit{Small}\,(x'-1)$, a contradiction.

The correction of the algorithm is proved. We discuss now the running time of the algorithm. Leaving apart temporarily the update of *nextt*(), the running time of Filter is proportional to v_t, where v_t is the number of elements w for which $\mathit{Small}\,(w)$ has smallest value t (these elements are discarded in step 12). Over all the executions of Filter, we obtain $O(v)$, where $v = \Sigma_{1 \leq t < n} v_t$. Now, v is in $O(n)$, since it counts the total number of elements w.

We need now to show that the update of *nextt*() in step 10 of Filter may be done in $O(1)$. For this, we see each strip s as a set $T(s)$ which has a representative element $r_{T(s)}$ such that $\mathit{nextt}(r_{T(S)})$ gives the successor of all the elements in the strip s (as insured by the definition of *nextt*()). Then it is sufficient to update $\mathit{nextt}(r_{T(S)})$ in order to update *nextt*() for all the elements in s. Now, notice that strips may be changed in two ways:

(1) the instruction Pop_R in *LR*-Search may perform deletions from some strip s. However, we keep the deleted elements in the set $T(s)$ and thus the representative $r_{T(s)}$, as well as the set $T(s)$, are unchanged by these deletions (although the strip itself is reduced).

(2) in step 10 of Filter, the element x becomes untrusty and then the strip immediately preceding x (if any), the strip formed by x alone, and the strip immediately following x (if any) may concatenate (altogether or only two of them, which have consecutive elements). Concatenations of strips imply unions of the corresponding sets.

In conclusion, updating the strings and being able to find the representative of each of them places us in the context of a Union-Find structure. The sets are the sets $T(s)$, containing the elements of s as well as all the elements previously in s and discarded from R by Pop_R. The unions between sets are given by the concatenations between strips, whereas the find operation for an element y seeks the representative element of the strip containing y. Again, we are in the particular case where unions always involve sets of consecutive integers (this is the case when two strips are concatenated). Thus, according to the result in [16], the implementation of unions and finds may be done in time linear with respect to the number of union and find operations.

Consequently, the operation Update(*nextt*()) in step 10 needs to concatenate if necessary two or three of the strips indicated in (2) and to update *nextt*() for their representative elements. Over all the executions of Filter, these operations are done in $O(n)$ and the running time of the algorithm is proved.∎

Other classes of common intervals

In [13], an algorithm is proposed for finding, in K signed permutations, all common intervals whose elements have the same sign inside each permutation. To solve this case, our *LR*-Search algorithm needs to (1) preprocess $\mathcal{P}$ to compute, for each t and k, the minimum $x_k^t > t$, such that t and x_k^t have different signs in P_k; and (2) stop to output intervals in the Filter procedure (step 4 in Algorithm 3) as soon as the first x with $x \geq \min\{x_k^t \mid 2 \leq k \leq K\}$ is found.

For a fixed k, task (1) is easily done in $O(n)$ by considering the elements of P_k in decreasing order, and remembering the sign changes. The global time required for this task is thus in $O(Kn)$, whereas the additional space may be limited to $O(n)$ by computing the $\min()$ values above progressively.

6.2 Maximal nested intervals

In [14], authors define a nested interval I to be *maximal* if it is not included in a nested interval of size $|I|+1$. In [8], an efficient algorithm is proposed to solve MaximalNested-ISP for $K = 2$. Not surprisingly, *LR*-Search works in this case too, with an appropriate filtering algorithm.

To this end, note that:

Claim 9. *A nested interval $(t..x)$ is maximal iff it satisfies the two following conditions:*

(a) $(x, x+1)$ *is a gap with* $x \in V_t$, *or* $x = y_t$

(b) $b_{t-1} < t-1$ *or* $B_{t-1} > x$.

Proof of Claim 9. By definition, $(t..x)$ is maximal iff it is nested but neither $(t..x+1)$ nor $(t-1..x)$ are nested. By Claim 5, this is equivalent to $x \in V_t$ but $x+1 \notin V_t$ and $x \notin V_{t-1}$ (or V_{t-1} is not defined). Now, $x \in V_t$ and $x+1 \notin V_t$ simultaneously hold iff property (a) in the claim is satisfied.

Moreover, $x \in V_t$ and $x \notin V_{t-1}$ (or V_{t-1} is not defined) iff, at the end of step 9 of **for**$_{t-1}$, either $t-1$ is not on top of L or x has been removed from R. Equivalently, $t-1 > b_{t-1}$ or $x < B_{t-1}$. The claim is proved.∎

Example 9. It is easy to see that in Figure 1, only intervals $(3..6)$ and $(1..3)$ satisfy these conditions. They are indeed the only maximal nested intervals of $\mathcal{P}$.

Then we have:

Theorem 7. *Algorithm LR-Search with settings b, B as for nested intervals and an appropriate Filter algorithm solves MaximalNested-ISP for K permutations in $O(Kn+N)$ time and $O(n)$ additional space, where N is the number of maximal nested intervals of $\mathcal{P}$.*

Proof of Theorem 7. According to Claim 9, Filter should output only the intervals that satisfy conditions (a) and (b). The latter condition is easy to test. The former one needs to find each gap, as well as y_t, in $O(1)$. For this, it is sufficient to compute and store, for each $r \in R$, the pointer $R^g(r)$ defined in Remark 5. Values $R^g(t+1)$ must be initialized immediately after *Push*$_{LR}(b_t, t+1)$, in step **for**$_t$ of *LR*-Search. They do not need to be updated.

Then it is sufficient to modify Filter in Algorithm 4 by replacing steps 10-15 with the following ones:

10: $x \leftarrow R^g(x^\top)$; Compute y_t
11: **while** $x \le y_t$ and ($b_{t-1} < t-1$ or $B_{t-1} > x$) **do**
12: Output the interval $(t..x)$
13: $W[t-1] \leftarrow x$ // *t passes down to $t-1$ its largest x such that $(t..x)$ is output*
14: $x \leftarrow R^g(next(x))$ //*or $x \leftarrow n+1$ if next(x) does not exist*
15: **end while**

Computing y_t in $O(1)$ is possible according to Remark 5. The other modifications aim at precisely selecting the elements with properties (a) and (b) in Claim 9. Notice that when a first value x not satisfying properties (a) and (b) is found in step 11, it is clear that no other subsequent value x' (necessarily $x' > x$) will satisfy properties (a) and (b). The resulting Filter procedure then finds all the maximal nested intervals, by Claim 9, runs in global time proportional to the number of output intervals, and globally uses $O(n)$ additional space.∎

6.3 Irreducible conserved intervals

In [6], authors define a conserved interval to be *irreducible* if it is not the union of smaller conserved intervals. They also give an efficient algorithm to solve IrreducibleConserved-ISP. Such an algorithm may also be obtained using *LR*-Search and the following easy result:

Claim 10. *Let $(t..x)$ be a conserved interval of $\mathcal{P}$. Then $(t..x)$ is irreducible iff*

$$x = \min\{h \mid t < h \text{ and } (t..h) \text{ is a conserved interval of } \mathcal{P}\}.$$

Proof of Claim 10. In [6] it is shown that two different irreducible intervals are either disjoint, or nested with different endpoints or else overlapping on one element. The conclusion follows.∎

We deduce:

Theorem 8. *Algorithm LR-Search, with settings b, B as for conserved intervals and a simplified Filter procedure, solves IrreducibleConserved-ISP for K permutations in $O(Kn)$ time and $O(n)$ additional space.*

Proof of Theorem 8. By Claim 10, it is sufficient to replace the **while** loop in the Filter procedure with an instruction that outputs $(t..x^\top)$.∎

7 Conclusion

The *LR*-stack we introduced in this paper is a simple data structure, of which we noted at least two advantages: it is powerful (we had two applications of it in this paper), and it is algorithmically efficient, since it makes use of the efficiency reached by the Union-Find-Delete algorithms.

Using *LR*-stacks, our algorithmic framework *LR*-Search succeeds in proposing a unique approach for dealing with common intervals and their subclasses. The computation of the interval candidates is driven by the *MinMax*-profile and the bounding functions, whose role is to guarantee that all interval candidates satisfy the content-related constraints. Afterwards, the Filter procedure chooses between the candidates those that satisfy the supplementary constraints defining a precise subclass.

All the algorithms resulting from this approach are as efficient as possible. They allowed us to prove the power and the flexibility of our approach. Among them, the algorithms searching for nested and maximal nested intervals of K permutations, with $K > 2$, solve previously unsolved problems.

References

[1] Stephen Alstrup, Inge Li Gørtz, Theis Rauhe, Mikkel Thorup, and Uri Zwick. Union-find with constant time deletions. In *Proceedings of ICALP 2005, LNCS*, volume 3580, pages 78–89, 2005.

[2] Sébastien Angibaud, Guillaume Fertin, and Irena Rusu. On the approximability of comparing genomes with duplicates. In *Proceedings of WALCOM, LNCS*, volume 4921, pages 34–45, 2008.

[3] Sébastien Angibaud, Guillaume Fertin, Irena Rusu, and Stéphane Vialette. How pseudo-boolean programming can help genome rearrangement distance computation. In *Proceedings of RECOMB-CG, LNCS*, volume 4205, pages 75–86, 2006.

[4] Michael Bender and Martin Farach-Colton. The LCA problem revisited. In *Proceedings of LATIN 2000, LNCS*, volume 1776, pages 88–94, 2000.

[5] Anne Bergeron, Cedric Chauve, Fabien de Montgolfier, and Mathieu Raffinot. Computing common intervals of k permutations, with applications to modular decomposition of graphs. In *Proceedings of ESA, LNCS*, volume 3669, pages 779–790, 2005.

[6] Anne Bergeron and Jens Stoye. On the similarity of sets of permutations and its applications to genome comparison. In *Procedings of COCOON, LNCS*, volume 2697, pages 68–79, 2003.

[7] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

[8] Guillaume Blin, David Faye, and Jens Stoye. Finding nested common intervals efficiently. *Journal of Computational Biology*, 17(9):1183–1194, 2010.

[9] Guillaume Blin and Romeo Rizzi. Conserved interval distance computation between non-trivial genomes. In *Proceedings of COCOON, LNCS*, volume 3595, pages 22–31, 2005.

[10] Gilles Didier. Common intervals of two sequences. In *Proceedings of WABI, LNCS*, volume 2812, pages 17–24, 2003.

[11] Michael Y. Galperin and Eugene V. Koonin. Who's your neighbor? new computational approaches for functional genomics. *Nature Biotechnology*, 18(6):609–613, 2000.

[12] Steffen Heber, Richard Mayr, and Jens Stoye. Common intervals of multiple permutations. *Algorithmica*, 60(2):175–206, 2011.

[13] Steffen Heber and Jens Stoye. Algorithms for finding gene clusters. In *Proceedings of WABI, LNCS*, volume 2149, pages 252–263, 2001.

[14] Rose Hoberman and Dannie Durand. The incompatible desiderata of gene cluster properties. In *Proceedings of RECOMB-CG, LNCS*, volume 3678, pages 73–87, 2005.

[15] Martijn Huynen, Berend Snel, Warren Lathe, and Peer Bork. Predicting protein function by genomic context: quantitative evaluation and qualitative inferences. *Genome Research*, 10(8):1204–1210, 2000.

[16] Alon Itai. Linear time restricted union/find, Manuscript, 2006. *http://www.cs.technion.ac.il/∼itai/Courses/ds2/lectures/UF/uf.pdf*.

[17] Warren C Lathe, Berend Snel, and Peer Bork. Gene context conservation of a higher order than operons. *Trends in Biochemical Sciences*, 25(10):474–479, 2000.

[18] Ross Overbeek, Michael Fonstein, Mark DSouza, Gordon D Pusch, and Natalia Maltsev. The use of gene clusters to infer functional coupling. *Proceedings of the National Academy of Sciences*, 96(6):2896–2901, 1999.

[19] Irena Rusu. New applications of interval generators to genome comparison. *Journal of Discrete Algorithms*, 10:123–139, 2012.

[20] Thomas Schmidt and Jens Stoye. Quadratic time algorithms for finding common intervals in two and more sequences. In *Proceedings of CPM, LNCS*, volume 3109, pages 347–358, 2004.

[21] Javier Tamames. Evolution of gene order conservation in prokaryotes. *Genome Biology*, 2(6):R0020, 2001.

[22] Javier Tamames, Georg Casari, Christos Ouzounis, and Alfonso Valencia. Conserved clusters of functionally related genes in two bacterial genomes. *Journal of Molecular Evolution*, 44(1):66–73, 1997.

[23] Takeaki Uno and Mutsunori Yagiura. Fast algorithms to enumerate all common intervals of two permutations. *Algorithmica*, 26:290–309, 2000.

[24] Christian Von Mering, Martijn Huynen, Daniel Jaeggi, Steffen Schmidt, Peer Bork, and Berend Snel. String: a database of predicted functional associations between proteins. *Nucleic Acids Research*, 31(1):258–261, 2003.